Quantum, Parallelism & Ownership

MATSUSHITA Yusuke — Igarashi & Suenaga Lab, KyotoU

Joint work with HIRATA Kengo (UEdinburgh) & WAKIZAKA Ryo (KyotoU)

Dec 19, 2024 – PL Joint Seminar @NII, Tokyo

About Me

At ACM PLDI 2022

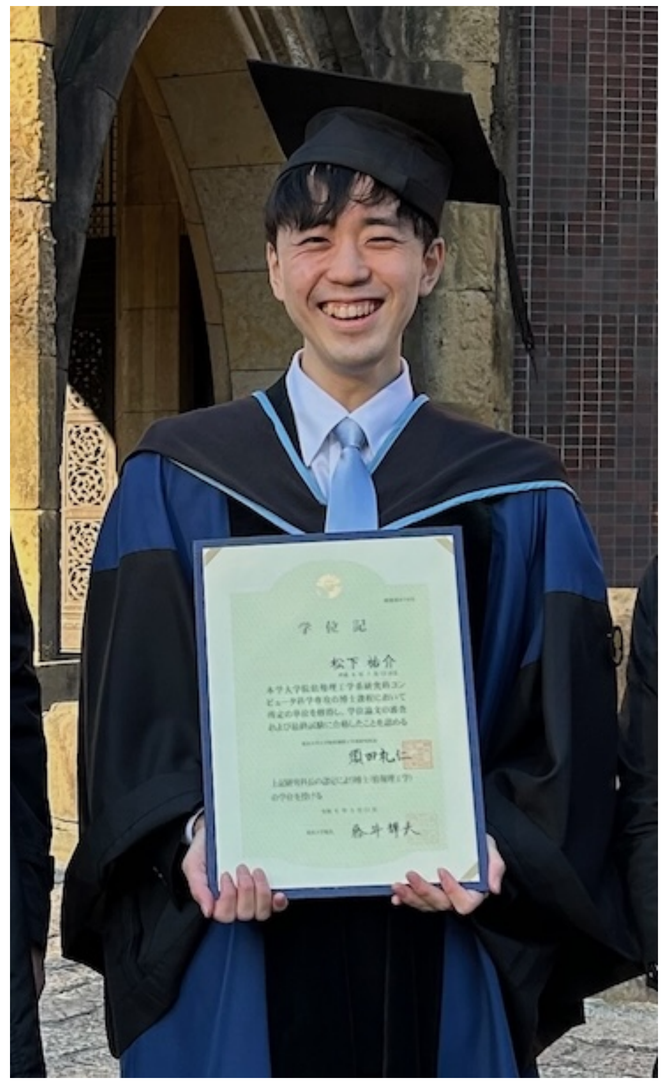
MATSUSHITA Yusuke

- Software scientist
  - Solid theories for real-world practice
- Loves & studies Rust
  - Rust is fun
- Loves music
  - Esp. improvisation

More about Me

Lecturer at IPA Security Camp 2024

S15 Rust Program Verification Seminar



Got a Ph.D. in 2024 at
the Dept. of Computer Science,
GS of IST, the University of Tokyo

Supervised by Prof. Naoki Kobayashi

Japan Bach Concours 2022
Gold Prize

Ricercar a 3, the Musical Offering

This Talk

- ✦ **Ongoing work: Concurrent quantum separation logic**
  - ▸ Focusing on qubit sharing for fine-grained parallelism
  - ▸ Joint work with HIRATA Kengo & WAKIZAKA Ryo
  - ▸ Presenting in TPSA & PLanQC '25 & Submitting to LICS '25
    - Questions & comments are super welcome!
- ✦ **New work: Linear Haskell × Rust-style borrows**
  - ▸ Rough idea, at an early stage
    - Looking for collaborators!

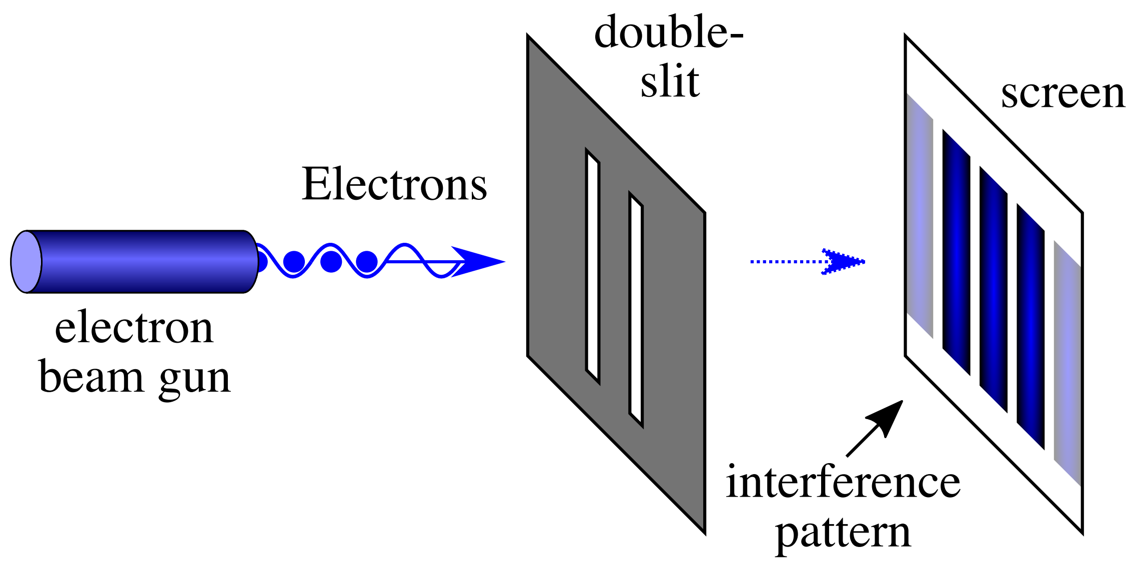
#1

Concurrent Quantum Separation Logic

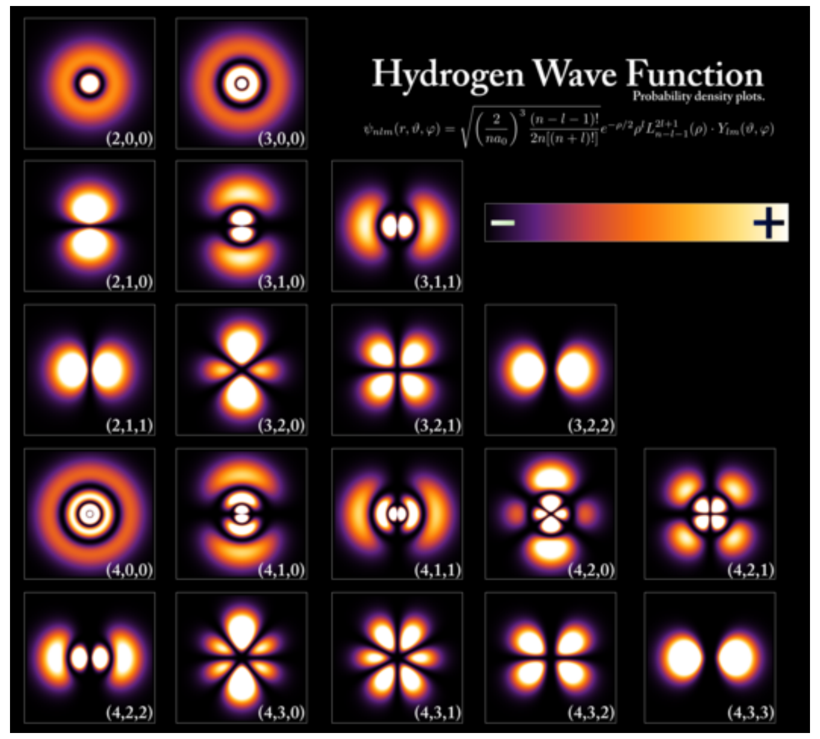
Ongoing joint work with HIRATA Kengo (UEdinburgh) & WAKIZAKA Ryo (KyotoU)

Quantum Mechanics Has Changed the World

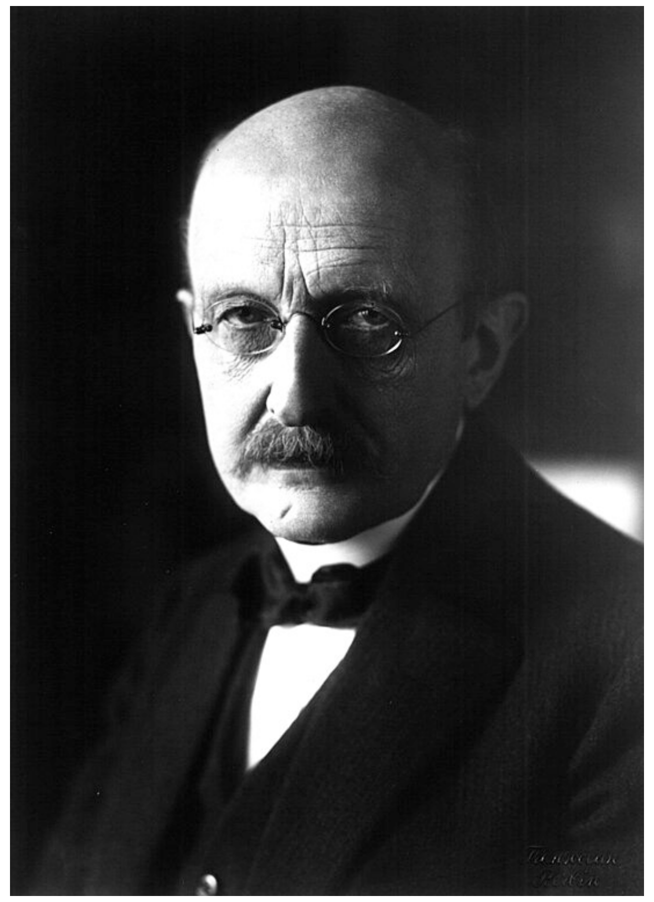
- Quantum mechanics is a foundation of modern science
  - Enabled computation about submicroscopic things
    - Molecules, atoms, photons, etc.
  - Key to modern physics, chemistry, biology, medicine, …



Double-slit experiment

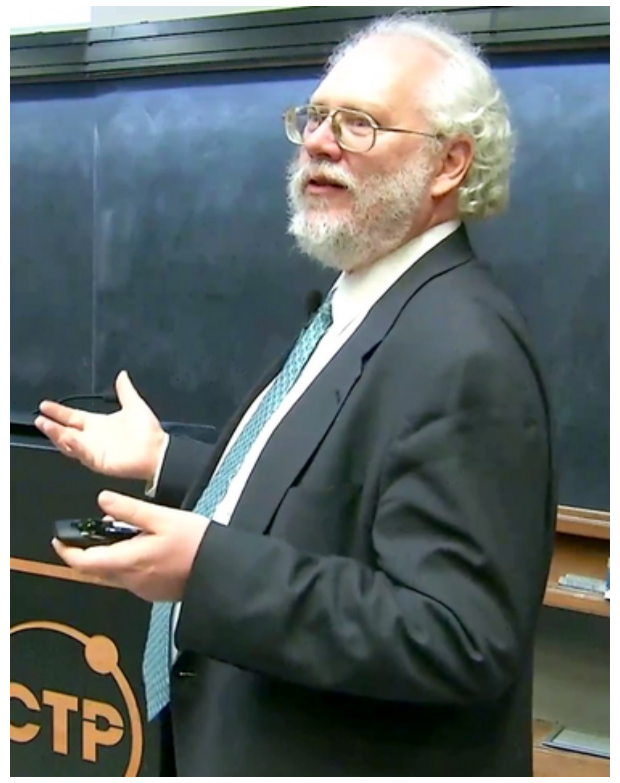


Hydrogen wave function


Max Planck

Quantum Computing Can Change the World?

- **Computing with quantum superposition**
  - E.g., Shor's algorithm for integer factoring
    - Quantum polynomial time, whereas classically only exponential algorithms are known
  - Possibly achieve "quantum supremacy"
- **May be practical in the near future?**
  - Challenge: Noise & decoherence
    - Tackled by hardware & error correction code



Peter Shor

IBM Quantum System One

What Is Quantum Computing?

- Uses quantum superposition of classical states
  - Can reason about multiple possibilities at once
    - Measurement probabilistically chooses a possibility

Classical State

$$0\cdots00,\ 0\cdots01,\ 0\cdots10,$$
$$0\cdots11,\ \ldots,\ 1\cdots11$$

2^k states, separately

Quantum State

$$\alpha_0 \left|0\cdots00\right\rangle + \alpha_1 \left|0\cdots01\right\rangle + \alpha_2 \left|0\cdots10\right\rangle +$$
$$\alpha_3 \left|0\cdots11\right\rangle + \cdots + \alpha_{2^k-1} \left|1\cdots11\right\rangle \qquad \alpha_0, \ldots, \alpha_{2^k-1} \in \mathbb{C}$$

Quantum superposition of 2^k states

Measurement

$|\alpha_0|^2$ $\qquad$ $|\alpha_1|^2$ $\qquad$ $|\alpha_{2^k-1}|^2$ $\qquad$ *Probability*

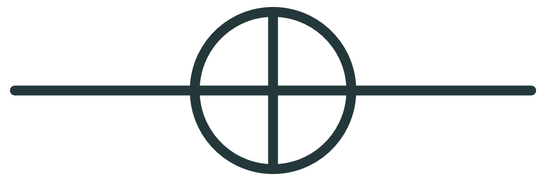
$$\left|0\cdots00\right\rangle \qquad \left|0\cdots01\right\rangle \qquad \cdots \qquad \left|1\cdots11\right\rangle$$

Quantum Logic Gates

- **Only a unitary matrix (or isometry) is allowed**
  - Linear map U that does not change the norm: $\left\| U \left| \psi \right\rangle \right\| = \left\| \left| \psi \right\rangle \right\|$
    - Invertible, and the inverse is the Hermitian adjoint: $U^{-1} = U^{\dagger}$

Various Gates

X gate



$$\mathrm{X} \triangleq \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\mathrm{X} \left| 0 \right\rangle = \left| 1 \right\rangle, \ \mathrm{X} \left| 1 \right\rangle = \left| 0 \right\rangle$$

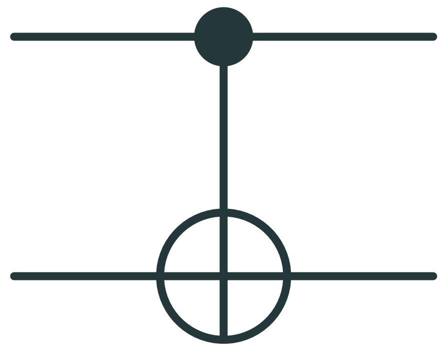
H gate

$$\mathrm{H} \triangleq \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathrm{H} \left| 0 \right\rangle = \left| + \right\rangle, \ \mathrm{H} \left| 1 \right\rangle = \left| - \right\rangle$$

$$\left| + \right\rangle \triangleq \frac{\left| 0 \right\rangle + \left| 1 \right\rangle}{\sqrt{2}}, \ \left| - \right\rangle \triangleq \frac{\left| 0 \right\rangle - \left| 1 \right\rangle}{\sqrt{2}}$$

CX gate

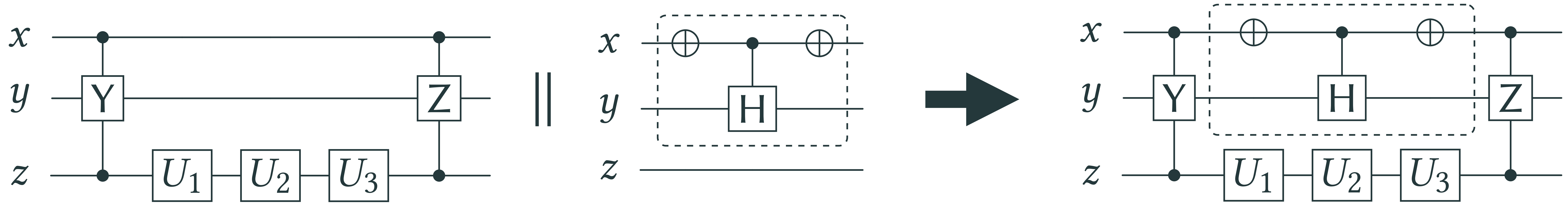


$$\mathrm{CX} \triangleq \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix}$$

$$\mathrm{CX} \left| 00 \right\rangle = \left| 00 \right\rangle, \ \mathrm{CX} \left| 01 \right\rangle = \left| 01 \right\rangle,$$
$$\mathrm{CX} \left| 10 \right\rangle = \left| 11 \right\rangle, \ \mathrm{CX} \left| 11 \right\rangle = \left| 10 \right\rangle$$

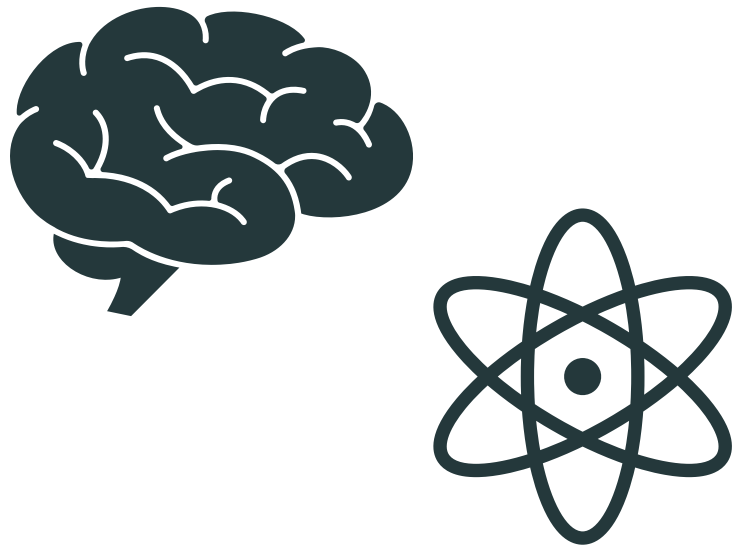
Topic: Parallelization of Quantum Programs

- **Parallelization of quantum programs is vital**
  - Reduces the depth of quantum circuits for runtime performance
  - Statically by quantum compilers or dynamically at runtime
- **Such parallelism is subject to tricky bugs**
  - Unexpected behaviors may occur only in some execution orders



Our Work: Concurrent Quantum Separation Logic

- **Concurrent quantum SL for fine-grained parallelism**
  - Separation Logic (SL) for modular reasoning
    - Separation ≈ Race Freedom, Disentanglement, Probabilistic independence
  - Key feature: Flexible sharing of qubits, for fine-grained parallelism
    - Allows semantically race-free parallel operations on the same qubits
    - Enjoys a form of completeness
    - New notion of ownership over quantum memory
  - Extension: Classical controls & Measurements

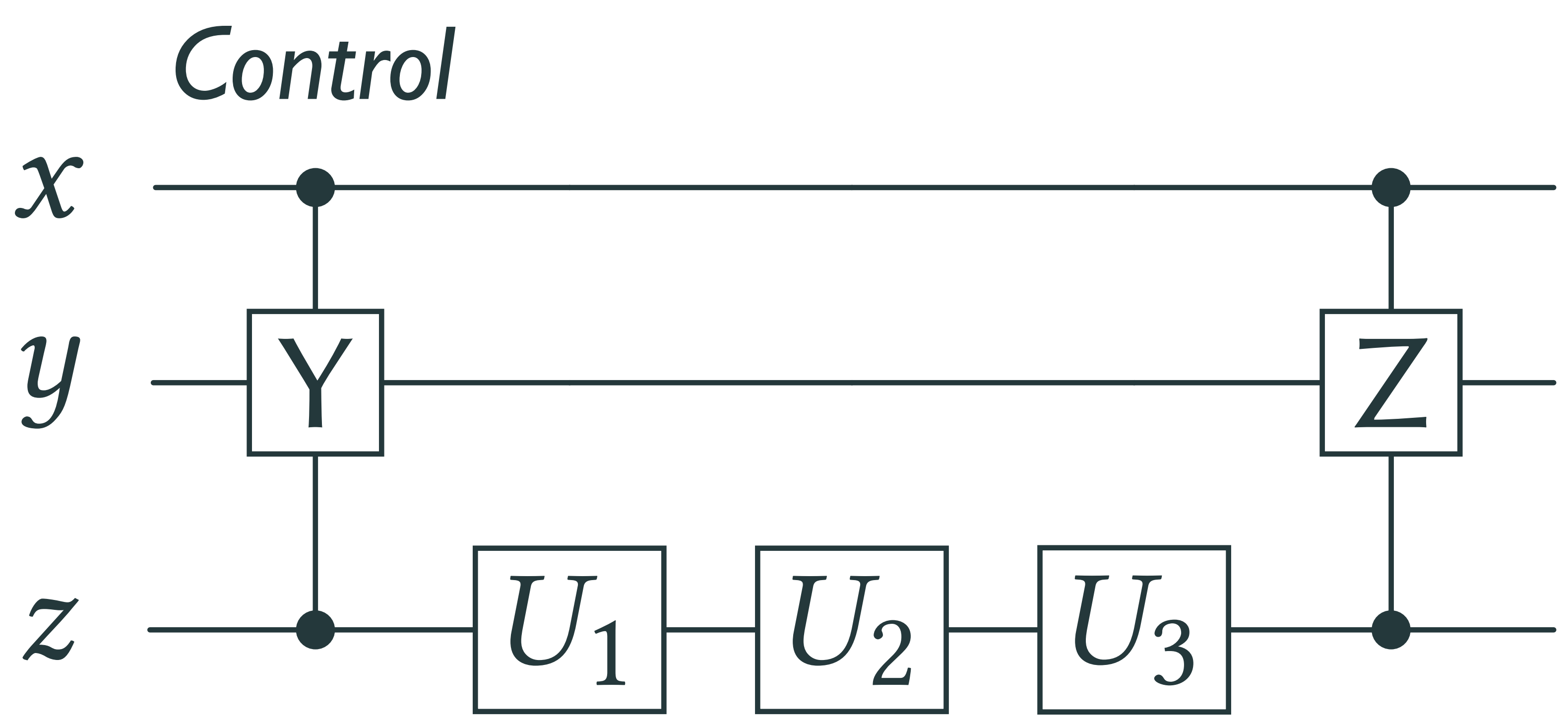


Non-Trivial Example of Fine-Grained Parallelism

Process 1 $\mathrm{CCY}[x, z, y];\ U_1[z];\ U_2[z];\ U_3[z];\ \mathrm{CCZ}[x, z, y]\ \ \|$

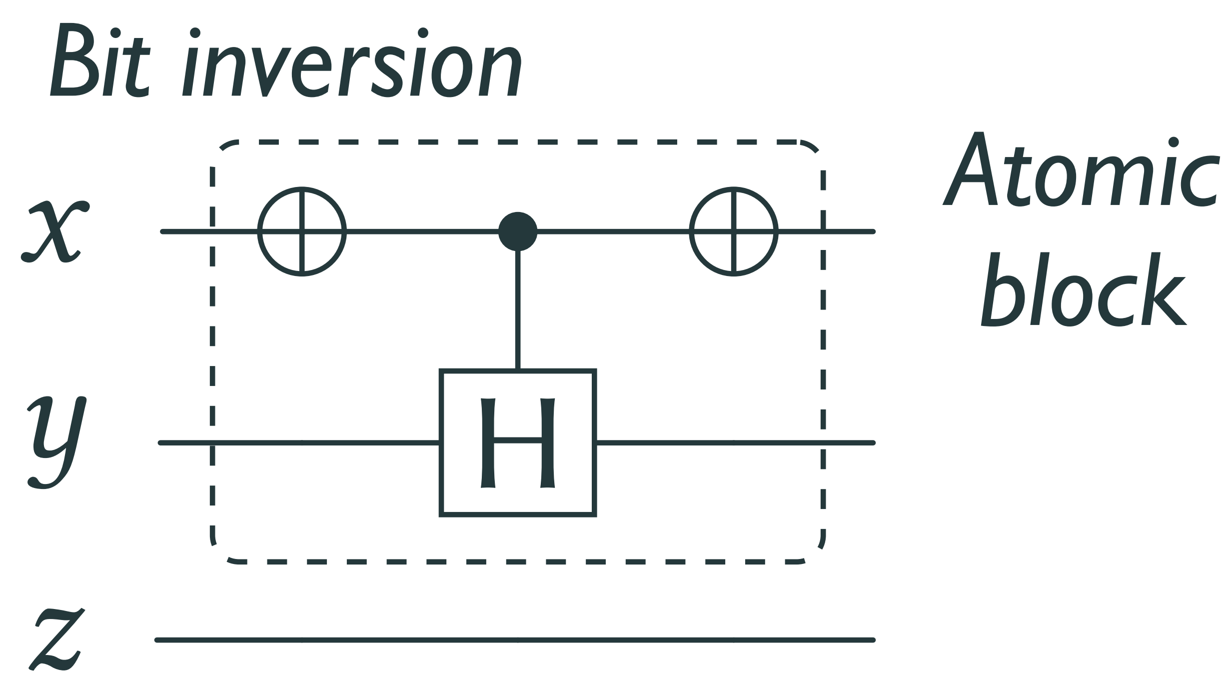
Process 2 $\texttt{atomic}\ \{\, \mathrm{X}[x];\ \mathrm{CH}[x, y];\ \mathrm{X}[x]\, \}$

Process 1

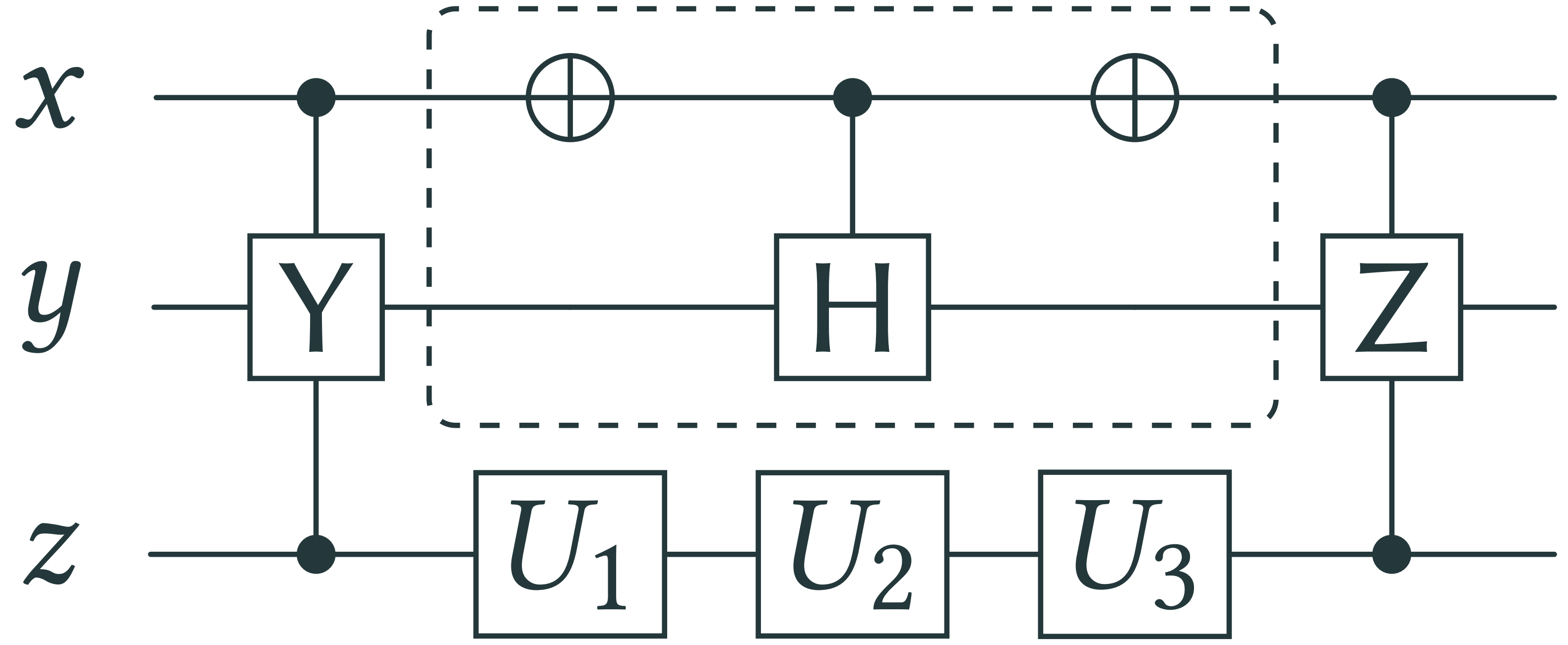


$\|$

Process 2



Clever scheduling

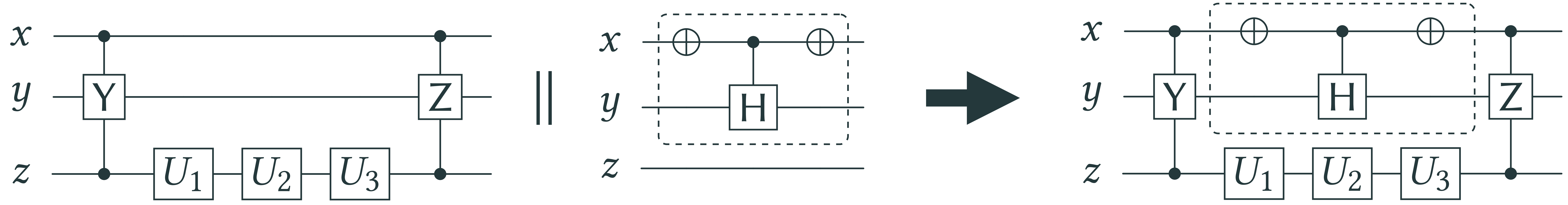


Question *Is this race-free?*

I.e., Is the result always the same regardless of the scheduling?

Can We Justify This Parallelism?

- **Challenge 1: Semantic race freedom**
  - Roughly, Process 1 & 2 write to y respectively only when x is 1 / 0
  - But quantum superposition: |0> & |1> can be mixed
- **Challenge 2: Treatment of atomicity**
  - Process 2 temporarily writes to x but reverts that in an atomic step



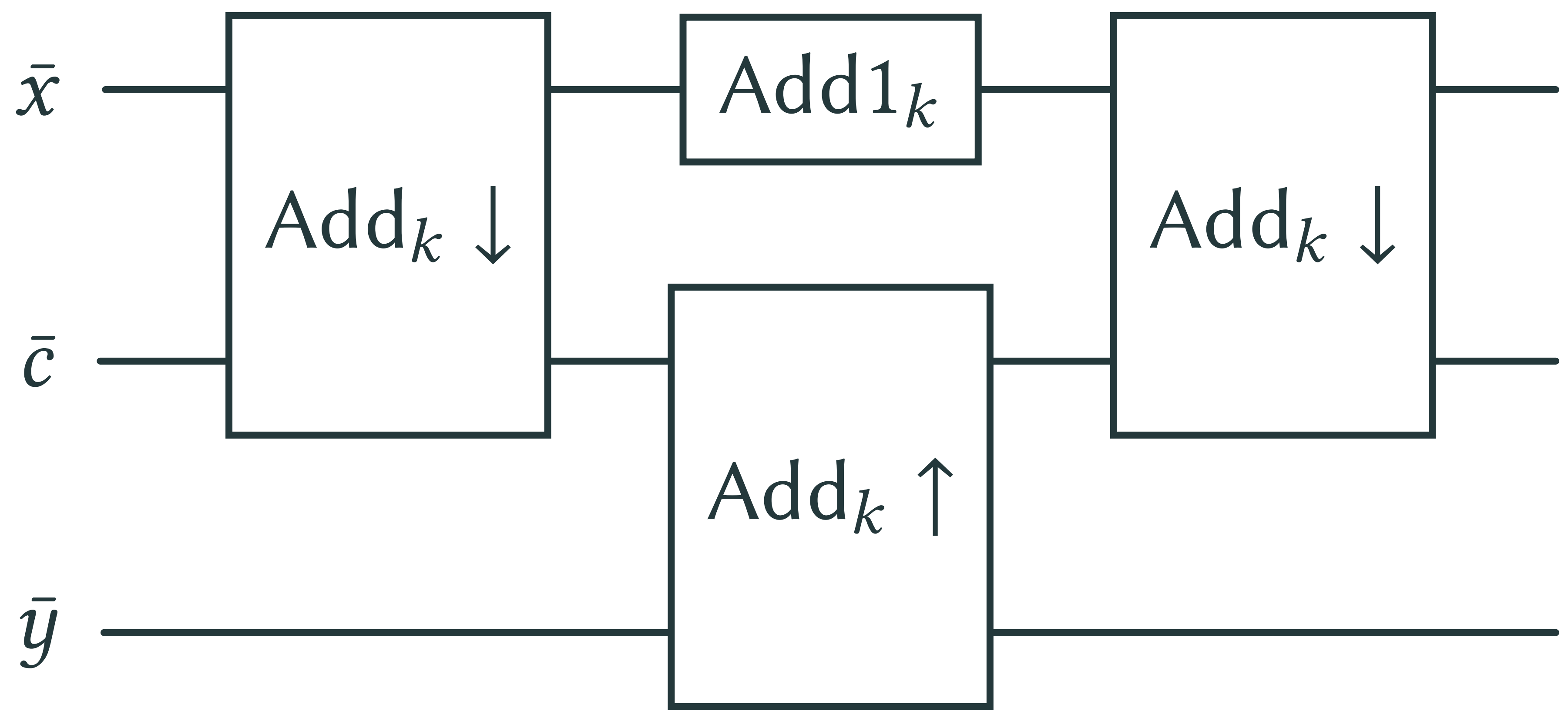
Another Interesting Example

- **Example of integer addition**
  - The result is unique thanks to the commutativity of the addition
  - Insight: Any classical reversible computing can be made quantum

$$\mathrm{Add}_k[\bar{x}, \bar{c}];\ \mathrm{Add1}[\bar{x}];\ \mathrm{Add}_k[\bar{x}, \bar{c}]$$
$$\|\ \mathrm{Add}_k[\bar{y}, \bar{c}]$$

Add_k : *Add a k-bit integer to another k-bit integer*

$\mathrm{Add1}_k$: *Increment a k-bit integer*



Basic Idea of Quantum Separation Logic

- **Quantum points-to token $\bar{x} \mapsto |\psi\rangle$**
  - The qubits $\bar{x} \in Qubit^k$ currently store the state vector $|\psi\rangle \in (\mathbb{C}^2)^{\otimes k}$
    - Not per single qubit, due to entanglement
  - Cf. Classical points-to token $\ell \mapsto v$
- **Separation ≈ Disjoint ownership & Disentanglement**

$$\bar{x} \mapsto |\psi\rangle \ * \ \bar{y} \mapsto |\phi\rangle \quad \equiv \quad (\bar{x}, \bar{y}) \mapsto |\psi\rangle\, |\phi\rangle$$

Tensor product

E.g.,

$$|0\rangle\, |1\rangle \ = \ |01\rangle$$

$$\bar{x} \mapsto |\psi\rangle \ \vdash \ \bar{x} \text{ are mutually disjoint}$$

Basic Rules of Quantum Separation Logic

- So far, quite like classical separation logic

$$\frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}}\ \textbf{Frame}$$

Uninvolved parts remain unchanged

$$\{\bar{x} \mapsto |\psi\rangle\}\ U[\bar{x}]\ \{\bar{x} \mapsto U|\psi\rangle\}$$

Gate

$$\frac{\{P\}\ e\ \{Q\} \qquad \{P'\}\ e'\ \{Q'\}}{\{P * P'\}\ e \parallel e'\ \{Q * Q'\}}\ \textbf{Parallel}$$

Ownership separation ensures safe parallel execution

$$\frac{\{P\}\ e\ \{Q\} \qquad \{Q\}\ e'\ \{R\}}{\{P\}\ e; e'\ \{R\}}$$

Seq

Our New Idea: Quantum Matrix Token

- **Quantum matrix token** $\bar{x} \mapsto^{i} U \mid \mathcal{S}$
  - Witness that the matrix U has been applied to the qubits
  - Has the ability to apply matrices in the set $\mathcal{S}$
  - i is the ID $\quad i \ ::= \ 1 \mid i.0 \mid i.1$
    - Given so that multiple matrix tokens can coexist

$$\frac{U \in \mathcal{S}}{\left\{\bar{x} \mapsto^{i} V \mid \mathcal{S}\right\} U[\bar{x}] \left\{\bar{x} \mapsto^{i} UV \mid \mathcal{S}\right\}} \quad \textbf{Gate'}$$

Borrows for Matrix Tokens

- We create matrix tokens by borrows or reborrows

$$\frac{\left\{\bar{x} \mapsto^{1} \mathsf{I} \mid \top\right\}\ e\ \left\{\bar{x} \mapsto^{1} U \mid \top\right\}}{\left\{\bar{x} \mapsto |\psi\rangle\right\}\ e\ \left\{\bar{x} \mapsto U\,|\psi\rangle\right\}}\ \textbf{Borrow}$$

Matrix commutativity $\mathcal{S} \leftrightarrow \mathcal{S}' \triangleq \forall A \in \mathcal{S}, B \in \mathcal{S}'.\ AB = BA$

$$\frac{\mathcal{S}_0, \mathcal{S}_1 \subseteq \mathcal{S} \qquad \mathcal{S}_0 \leftrightarrow \mathcal{S}_1 \qquad \left\{\bar{x} \mapsto^{i.0} \mathsf{I} \mid \mathcal{S}_0\ *\ \bar{x} \mapsto^{i.1} \mathsf{I} \mid \mathcal{S}_1\right\}\ e\ \left\{\bar{x} \mapsto^{i.0} U_0 \mid \mathcal{S}_0\ *\ \bar{x} \mapsto^{i.1} U_1 \mid \mathcal{S}_1\right\}}{\left\{\bar{x} \mapsto^{i} V \mid \mathcal{S}\right\}\ e\ \left\{\bar{x} \mapsto^{i} U_1 U_0 V \mid \mathcal{S}\right\}}\ \textbf{Reborrow}$$

Promotion by Atomicity

- **Promote a matrix token into a points-to by atomicity**
  - Kind of inverse of the frame rule, very subtle

Ownership exclusion

$$\frac{e \text{ is atomic} \qquad P : \mathrm{out}\ \bar{x} \qquad U \in \mathcal{S} \qquad \forall\, |\psi\rangle\, .\ \{\bar{x} \mapsto |\psi\rangle \ast P\}\ e\ \{\bar{x} \mapsto U\,|\psi\rangle \ast Q\}}{\{\bar{x} \mapsto^{i} V \mid \mathcal{S} \ast P\}\ e\ \{\bar{x} \mapsto^{i} UV \mid \mathcal{S} \ast Q\}}\ \textbf{Promote}$$

$$\frac{\{P\}\ e\ \{Q\}}{\{P\}\ \mathtt{atomic}\,\{e\}\ \{Q\}}\ \textbf{Atomic}$$

$\mathtt{atomic}\,\{e\}$ is atomic

We Enjoy Completeness!

Completeness Theorem

If the resulting matrix of a program e is uniquely U, then our separation logic can prove the following:

$$\forall\, |\psi\rangle\, .\; \left\{ \bar{x} \mapsto |\psi\rangle \right\}\, e\, \left\{ \bar{x} \mapsto U\, |\psi\rangle \right\}$$

Here, we take the program e from the following fragment:

$$e \quad ::= \quad U[\bar{x}] \mid e; e' \mid e \,\|\, e' \mid \texttt{atomic}\,\{e\}$$

We also assume the oracles for membership $U \in \mathcal{S}$, inclusion $\mathcal{S} \subseteq \mathcal{S}'$, and commutativity $\mathcal{S} \leftrightarrow \mathcal{S}'$

Proof of the Completeness

Key Lemma on Parallel Execution

If $e \parallel e'$ has a unique result, both e and e' have a unique result, each AC of e commutes with each AC of e' *AC = Atomic component*

$\because$ Since all matrices are invertible, uniqueness can be discussed locally

Proof of the Completeness

By proving the following by structural induction over e:

If e has a unique result U, then letting $\mathcal{S}$ be the set of e's ACs, our logic can prove, for any i: $\left\{\bar{x} \mapsto^{i} \mathrm{I} \mid \mathcal{S}\right\}\ e\ \left\{\bar{x} \mapsto^{i} U \mid \mathcal{S}\right\}$

Discharging Queries on Matrix Sets

We assume oracles on complex number arithmetic

- **Queries are decidable for finite sets $\mathcal{S}$**
  - Our completeness proof uses only a finite set of ACs
- **We can also consider the vector subspaces for $\mathcal{S}$**
  - Matrices form a vector space, and commutativity is well-behaved
    - If $\mathcal{S} \leftrightarrow \mathcal{S}'$, then that extends to the linear spans: $\operatorname{span} \mathcal{S} \leftrightarrow \operatorname{span} \mathcal{S}'$
  - The vector space of matrices is finite-dimensional, we can always take a finite (bounded) number of bases for $\mathcal{S}$, yay!
    - The membership, inclusion, and commutativity queries can be answered in terms of the bases (thanks linear algebra!)

More Efficient Answers to Matrix Set Queries?

- **Want to have a sophisticated proof system for answering queries on matrix sets efficiently**
  - Hopefully, it will be complete over some fragment
  - Hopefully, we can even design an efficient decision algorithm
  - The following fragment seems to suffice in practice

$$\mathcal{S} \;::=\; \top \mid \mathbb{C}\,\mathrm{I} \mid \mathcal{S} \otimes \mathcal{S}' \mid \mathcal{S} \oplus_{\mathcal{W}} \mathcal{S}'$$

Extension to qalloc / qfree

- **Allocating a fresh qubit / Deallocate a qubit**
  - Guaranteed to be initialized / Obliged to initialize to |0>
    - Can be naturally reasoned by points-to tokens

$$\frac{\forall x.\ \{x \mapsto |0\rangle * P\}\ e\ \{Q\}}{\{P\}\ \mathtt{let}\ x = \mathtt{qalloc}\ \mathtt{in}\ e\ \{Q\}}$$ **Qalloc**

$$\{x \mapsto |0\rangle\}\ \mathtt{qfree}\ x\ \{\mathrm{emp}\}$$ **Qfree**

Incompleteness under qalloc

- **Our logic is incomplete in the presence of qalloc**
  - Initialization to |0> makes more programs have a unique result
    - Even when matrix commutativity does not hold

Counterexample

$$\texttt{let}\ x = \texttt{qalloc}\ \texttt{in}\ \ \texttt{let}\ y = \texttt{qalloc}\ \texttt{in}\ \ \texttt{let}\ z = \texttt{qalloc}\ \texttt{in}$$
$$\mathrm{Add1}_3[x, y, z] \ \parallel \ \mathrm{Add1}_2[y, z]$$

Both order of execution gives $(x, y, z) \mapsto |010\rangle$

But the matrices $\mathrm{Add1}_3,\ I \otimes \mathrm{Add1}_2$ *do not commute*

Related Work

- **Quantum SLs** [Zhou+ LICS '21, Le+ POPL '22, Su+ '24]
  - Separation ≈ No sharing of qubits & Disentanglement
  - Supported neither concurrency nor sharing of qubits
    - Concurrency might be safely supported, but not discussed well
    - But sharing of qubits is fundamentally difficult

TODOs & Future Work

- Explanation, design of reasoning rules, proofs
- Extension to classical controls & measurements
  - Use Outcome Logic [Zilberstein+ '23] to model probabilistic choices
    - $P \oplus_p Q$: P by probability p, Q by probability (1 - p)
- Case studies of more practical examples
- Automated quantum program parallelization

#2

Linear Haskell × Rust-Style Borrows

New work at an early stage

Linear Haskell & Rust

- **Linear Haskell [Bernardy+ POPL ’18]**
  - Linear types [Wadler ’90] in GHC Haskell
  - Highly useful for achieving performative computation
    - Can encapsulate destructive updates into pure APIs under linearity
- **Rust [Matsakis & Klock ’15]**
  - Systems programming language with strong ownership types
  - Key feature: Borrows by lifetimes (&α `mut` T, &α T, …)
    - No need for direct communications in returning ownership

Proposal: Linear Haskell × Rust-Style Borrows

- **Rust-style borrows in Linear Haskell**
  - Can be provided as libraries for real-world GHC
  - Implementation: Just use unsafePerformIO etc.
    - Key challenge: Pure APIs encapsulating destructive updates
  - Haskell's high-level reasoning × Rust-like safe pointer manipulation
    - Can enjoy Haskell's data types, higher-order functions, lazy evaluation, etc.
    - Can enjoy flexible, efficient, and safe pointer manipulation, as in Rust

ST Monad in Haskell

- **ST Monad encapsulates destructive updates into purity**

Stateful computation *Pure value*

```
runST : (∀s. ST s a) -> a
```

Fresh memory region

Conceptually, the state monad over a fresh memory region s

Example

```
newSTArray : ST s (STArray s a)
readSTArray : STArray s a -> Int -> ST s a
writeSTArray : STArray s a -> Int -> a -> ST s ()
```

Linear Haskell

- **Linear arrow type: `a –o b`** *Written as a %1 –> b in GHC*
  - A function that consumes the argument exactly once
    - More precisely: For `f : a –o b`, if `f x : b` is consumed exactly once, the argument `x : a` is consumed exactly once
  - No need for ST monad
    - Easier to write & More chances for parallelism

Example

Unrestricted

```
newLArray : (LArray a –o Ur b) –o Ur b
readLArray : LArray a –o Int –> (LArray a, Ur a)
writeLArray : LArray a –o Int –> a –> LArray a
```

Linearity Witnesses

- **Linearity witness `Linearly`**
  - Witness that the result of the current computation is used linearly
    - Extensive library linear-witness by ISHII Hiromi
    - Linear Constraints [Spiwack+ ICFP '22] for even better interfaces

```
newLArray :
    (LArray a ⊸ Ur b) ⊸ Ur b
```

➡

```
newLArray : Linearly ⊸ LArray a
```

Or

```
newLArray : Linearly =o LArray a
```

```
linearly : (Linearly ⊸ Ur a) ⊸ Ur a
```

```
dup : Linearly ⊸ (Linearly, Linearly)    consume : Linearly ⊸ ()
```

Why Purity Matters?

- More predictable behaviors by referential transparency
  - Under lazy evaluation, concurrency, …
- Enables various optimizations
  - Fusion transformation, …

Example

```
map g (map f xs)    ➡    map (g . f) xs
```

Applications of f and g are swapped

Key Observation: Moves vs. Borrows

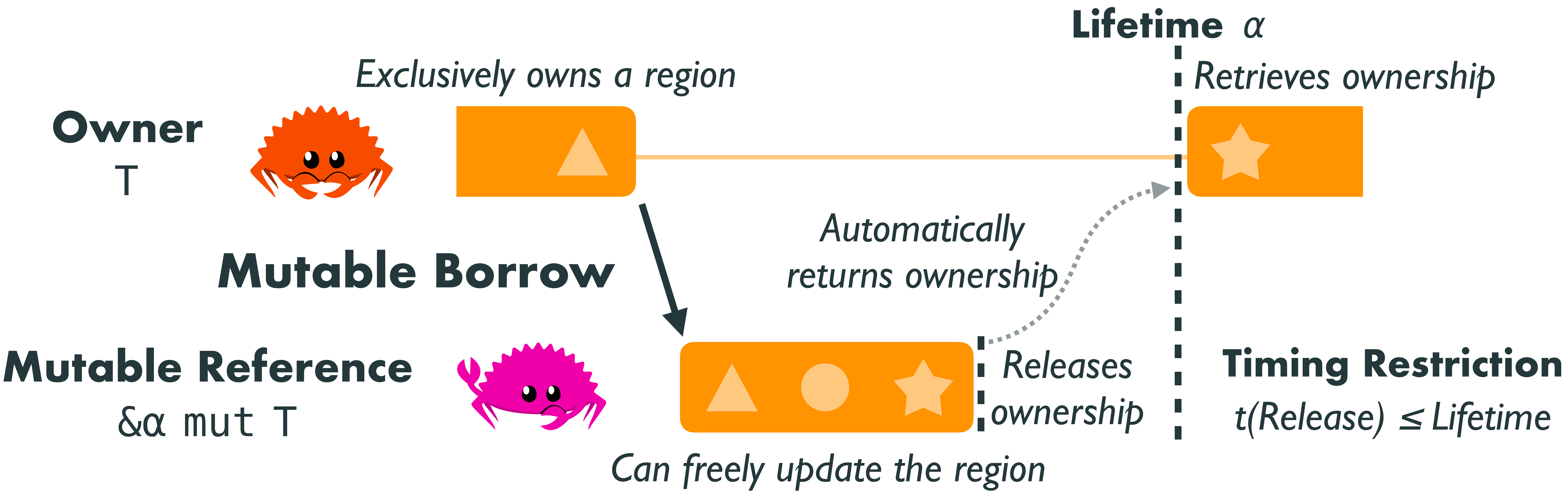
- Linear Haskell: Need to move the accessed data around
- Rust: Access by borrowing

```
readLArray : LArray a -o Int -> (LArray a, Ur a)
writeLArray : LArray a -o Int -> a -> LArray a
```

```
fn index<α,T>(v : &α Vec<T>, i : uint) -> &α T
fn index_mut<α,T>(v : &α mut Vec<T>, i : uint)
                                             -> &α mut T
```

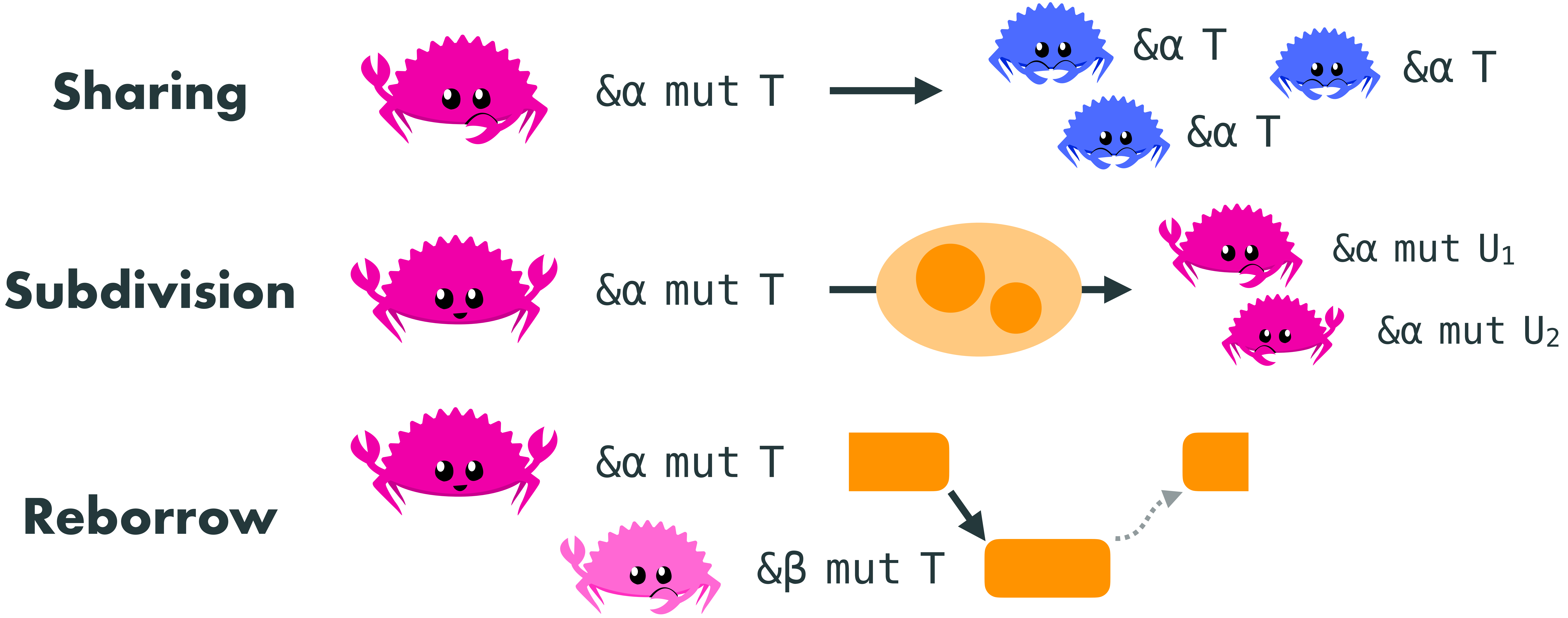
Rust's Borrows

- Temporarily borrow ownership
  - No direct communications is needed when releasing ownership



Rust's Operations on Borrows

- Various high-level operations on borrows are provided



Rust's Borrow Checking

- ✦ **Automatic static checking on borrows**
  - ▸ Esp. the timing restriction *t(Release)* ≤ *Lifetime*
- ✦ **Actively evolving over time**
  - ▸ Older (–2018) — Scope-based, lexical lifetimes
    - *t(Release)* is the end of the scope
  - ▸ Now — Non-lexical lifetimes by Niko Matsakis
    - *t(Release)* is inferred by liveness analysis
  - ▸ Future? — "Borrow checker within"
    - More info in types, self-borrows supported

Niko Matsakis

His blog "baby steps"

```
fn new_widget(&self, name: String) -> Widget {
    let name_suffix: &'name str = &name[3..];
                         // —-- borrowed from "name"
    let model_prefix: &'self.model str = &self.model[..2];
                          // ———-- borrowed from "self.model"
}
```

Simple Approach to Borrows in Linear Haskell

- Use a state monad over the lender

```
borrow : Linearly -o a -o ∃ α.    Fresh lifetime
           (MutBor α a, Lend α, Ur (Lend α -o a))
           Mutable borrower  Lender  Retrieve the borrowed object

BorST α a  ≜  Lend α -o (Lend α, a)   State monad over the lender

swap : MutBor α a -o a -o BorST α (a, MutBor α a)

consume : MutBor α a -o ()   Mutable borrowers can be released any time

indexMut : MutBor α (BArray a) -o Int ->
                                   BorST α (MutBor α a)
                          Mutable borrowers can be subdivided
```

Correctness

- **Memory safety**
  - By ensuring disjointness of mutable references
    - We can think of "logical paths" instead of physical addresses
      - E.g., Paths x.0, x.1.0, x.1.1 are disjoint
- **Purity**
  - By modeling `Lend` α with the store-passing style
    - C.f., how ST's purity is proved [Timany+ POPL '17, Jacobs+ OOPSLA '22]
  - Prove bisimulation between the non-updating computation model
    - Similar to the correctness proof of the Linear Haskell paper

Challenges & Future Work

- **Reborrows?**
  - Reborrowers involve multiple lenders, so APIs are a bit more involved
- **Smooth reasoning about lifetimes?**
  - Especially when handling multiple lifetimes
- **Parallel accesses to one lender?**
  - Somehow share `Lend α` between processes?
  - Use RustBelt-like APIs with lifetime tokens? How to ensure purity?
- **Abstractions like lenses?**

Thank you!